TRABAJO PRACTICO 2:

FILTRO DE IMAGENES

PROFESOR: Dario Turco

Integrantes:

Federico Cerocchi

Manuel Gudiño

Matheo Maidana

Lucas Santander

Curso: 6AO
Materia: Programación Sobre Redes
Profesor: Darío Turco
Alumnos: Federico Cerocchi, Manuel Gudiño, Matheo Maidana y Lucas Santander

Introducción

El presente informe tiene como objetivo describir las metas del segundo proyecto de la materia, Programación sobre redes, una descripción detallada de funcionamiento del programa, una explicación de los experimentos realizados, y una resolución con una conclusión que cierre nuestros pensamientos en la experiencia.

Objetivos del trabajo

Nuestra investigación se enfoca en el desarrollo de un programa que permita recibir, procesar y crear imágenes mediante la implementación de una serie de filtros, similares a los utilizados en plataformas populares como Instagram. Este programa brindará al usuario la posibilidad de seleccionar el filtro deseado, el número de threads a utilizar, la imagen de entrada y los parámetros correspondientes al filtro seleccionado.

El propósito fundamental de este proyecto es explorar las capacidades de procesamiento de imágenes digitales, así como desarrollar habilidades prácticas en la implementación de algoritmos de visión por computadora, sumado al campo de la utilización de los threads de los procesadores.

¿Qué se aprenderá con este trabajo?

Se abordarán los fundamentos teóricos y técnicos necesarios para comprender y aplicar los conceptos de procesamiento de imágenes digitales y threads. Se analizarán diferentes técnicas y algoritmos utilizados en el ámbito del Computer Vision, y se implementarán mediante la utilización de un lenguaje de programación adecuado para nosotros, en este caso C++.

Asimismo, se realizarán pruebas exhaustivas del programa desarrollado, evaluando su eficiencia y rendimiento bajo distintas condiciones y escenarios, como la cantidad de threads usados, la cantidad de imágenes procesadas, o el tamaño de las mismas. Con estos resultados, se pretende demostrar la viabilidad y utilidad de la solución propuesta, así como identificar posibles mejoras o ampliaciones para futuros trabajos en el campo de los procesadores o, de forma más exacta, los threads.

Curso: 6AO
Materia: Programación Sobre Redes
Profesor: Darío Turco
Alumnos: Federico Cerocchi, Manuel Gudiño, Matheo Maidana y Lucas Santander

Filtros

“Pixel to Pixel”

- Brightness: Se reciben cuatro parámetros: la imagen, el porcentaje de brillo al que se le quiere someter. Se itera sobre cada píxel, se calcula el brillo y luego se multiplica a los valores RGB por el mismo, poniendo como máximo valor 255.

- blackWhite: Este filtro recibe como único parámetro una imágen, indicada por el usuario. Luego, se itera por cada píxel y se obtienen sus valores RGB mediante el método getPixel. Por último, se calcula el promedio de dicho RGB (sumándolos y dividiendo por 3) y se settea este promedio como valor de los 3 colores del mismo píxel, con el método setPixel.

- Contrast: Iteramos sobre todos los pixeles y obtenemos su valor RGB. Realizamos la fórmula especificada utilizando el input de contraste ingresado por el usuario, y luego nos aseguramos de que cada valor no sea menor a 0 o mayor a 255. De serlo, regular sus valores para que estén dentro del rango [0 , 255]. Setteamos RGB con dichos valores.

- Shades: Iteramos sobre cada píxel y obtenemos su RGB. Sacamos el valor del rango, que se calcula dividiendo a 255 por el input del usuario restado en 1. Luego se obtiene un promedio del RGB, para luego dividirlo por el rango y multiplicarlo nuevamente por el mismo para hacerlo entero. Por último, se settea el RGB con el resultado de esta cuenta.

- Merge: Se reciben dos imágenes como input del usuario. Deben de tener la misma resolución para que el filtro funcione. Se obtienen sus RGB y se calcula para red, green y blue, un valor nuevo que combina cierta parte de cada imagen, dependiendo del input del usuario que indica esto mismo.

“Convolution filters”

- BoxBlur: Se itera sobre cada píxel, se suman los valores de sus 9 píxeles vecinos y se calcula un promedio entre ellos (por separado). Luego, se settea el valor R con el promedio de R, el valor G con el promedio de G y el valor de B con el promedio de B, en una imagen temporal. Ya habiendo hecho todo esto, podemos pasar el resultado, construido en la imagen temporal, a la imagen original.

- Sharpen: Se itera sobre cada píxel, y mediante la matriz especificada se aplica el efecto del kernel a los píxeles vecinos. Luego, se asigna el valor del coeficiente según la posición del kernel. Ya habiendo hecho esto en cada píxel, se modifica a los valores para encajar en el rango [0 , 255] y se setea el nuevo RGB en una imagen temporal. Cuando todos los píxeles ya han sido recorridos, se reemplaza la imagen original por la temporal.

- edgeDetection: Primero se aplica el filtro blackWhite y luego el blur a la imagen original. Luego, mediante el sobel, se calcula la convolución para el píxel actual y la contribución del píxel al valor de convolución. Por último, se calcula la magnitud del gradiente y se establece el valor de intensidad obtenido en el RGB del píxel. Nuevamente, se reemplaza la imagen original por una temporal que se va modificando en cada iteración.

Dificultades atravesadas

- Al comienzo tuvimos problemas para entender el método getPixel, pero resultó ser un simple error de sintaxis (agregar llaves de apertura y cierre a los bucles for).
- Para todos los filtros de convolución tuvimos que tomar una precaución muy importante, no hacer referencia a píxeles que no existen. Al recorrer los píxeles vecinos en los píxeles que de por sí están en el borde de la imagen, nos encontramos con un error de address boundary, al intentar referenciar al píxel -1, -1 por ejemplo. Lo mismo con los píxeles finales. Para evitar esto, simplemente modificamos el bucle que recorre los píxeles, empezando a iterar en 1 en vez de 0 y hasta img.<dimensión> -1 en vez de img.<dimensión>.
- También tuvimos algunas dificultades para hacer correctamente la división de píxeles por thread. Decidimos incluir los píxeles restantes, obtenidos sacando el resto de la división del height hechas en los multi-thread, y colocarlos en el último thread para que este también los itere. Al principio no nos dimos cuenta de este problema, por lo que las imágenes quedaban cortadas y dejando un espacio sin el filtro aplicado.
- Por último, al realizar multi-thread edgeDetection por primera vez, aplicamos el filtro blackWhite y el boxBlur single-thread, para luego hacer multi-thread en el cálculo particular de este filtro. Luego, después de varios intentos, pudimos organizar el código para reutilizar las funciones multi-thread de iteración de las funciones blackWhite y boxBlur, por lo que no tuvimos que repetir esa lógica y el resultado quedó más limpio, además de haber optimizado la solución.

Curso: 6AO
Materia: Programación Sobre Redes
Profesor: Darío Turco
Alumnos: Federico Cerocchi, Manuel Gudiño, Matheo Maidana y Lucas Santander

Versiones multi-thread:

Procedimiento para encarar las funciones multi-thread:
Decidimos dividir las imágenes dependiendo la cantidad de threads:
(int pixeles = img.height / n;) y que cada uno de ellos se encargue de aplicar el filtro a un área específica.

Definimos un vector 'threads', luego, dividimos la cantidad de píxeles por la cantidad de threads que el usuario ingresó como parámetro, para así obtener la cantidad de píxeles de la imagen a la que le aplicará el filtro cada thread.
Luego se itera hasta alcanzar la cantidad de threads ingresados, dentro de este bucle, se agrega al vector 'threads' un llamado a la función que setea los pixeles en la imagen junto con los parámetros correspondientes, entre ellos el parámetro 'resto': Si es la última iteración, el parámetro resto será (int resto = img.height % n;) en el resto de iteraciones, el resto valdrá 0. Hacemos esto para evitar tener cantidades de píxeles decimales.
Finalmente, se itera sobre el vector 'threads' donde se ejecuta 'thread.join()' por cada iteración, está función se asegura de que cada thread termine su ejecución antes de pasar a la siguiente.

Experimento:

Para nuestro experimento realizamos un filtro single-thread llamado gradiente. Este filtro recibe como input del usuario un parámetro denominado "dimensión", que puede ser 1 o 0. De ser 1, la dimensión elegida es height, y de lo contrario, es width. Ya en el filtro, se chequea cúal es el caso para realizar un procedimiento o el otro.

Este filtro divide la dimensión elegida de la imagen por 255. De esta manera, podemos saber cuántas veces entra 255 en la cantidad de píxeles que conforman dicha dimensión. La idea es que cada cierto intervalo de píxeles, vaya cambiando la intensidad de red en el RGB del píxel. G y B quedan iguales, pero red empieza en 0. A medida que el for que itera sobre los píxeles de la dimensión avanza, la variable que guarda el valor de red aumenta en 1, dependiendo de la cantidad de píxeles totales.

Si la imagen tiene 510 píxeles en cierta dimensión, cada dos iteraciones red aumenta en uno, de manera que su valor empieza en 0 y termina en 255

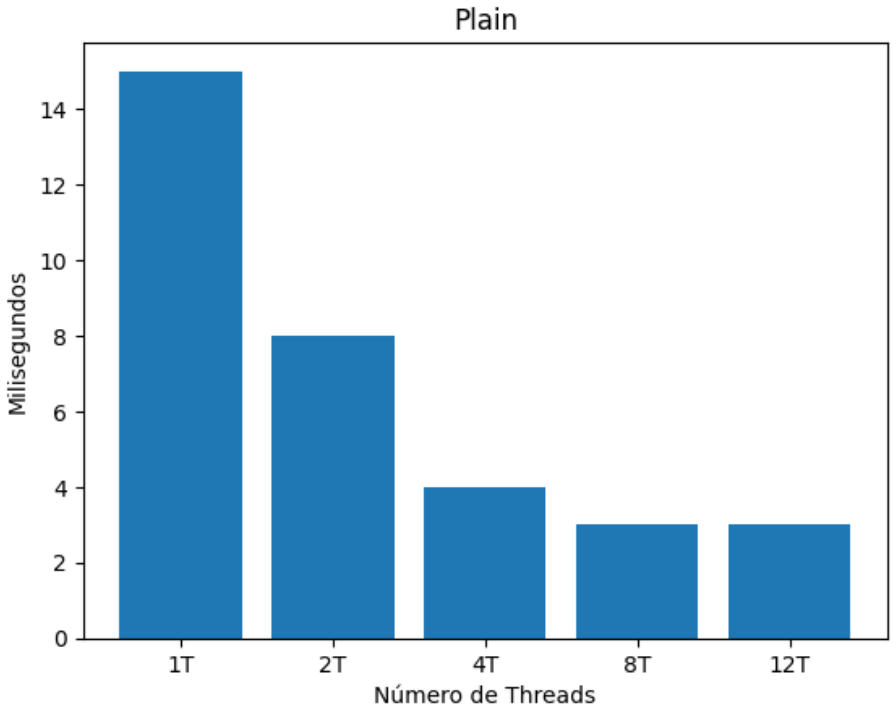
aproximadamente. En el 99% de los casos el valor de la división tiene resto, por lo que simplemente el resto se ignora y se coloca con un valor de 255 en red, considerando que el gradiente ya alcanzó su límite previamente.

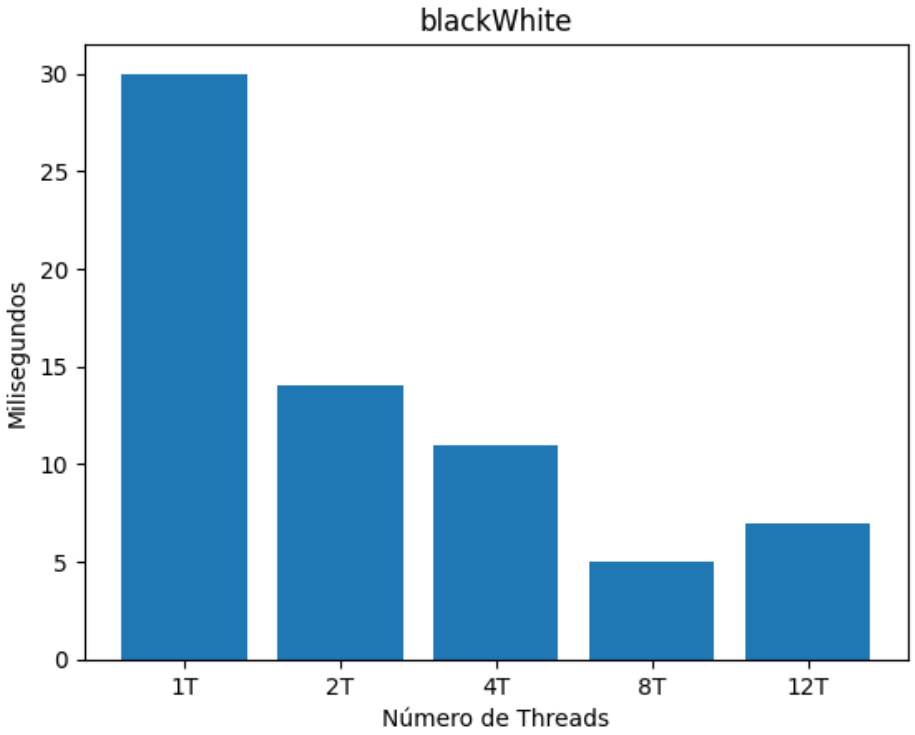
Si la imagen tiene menos de 255 píxeles en la dimensión elegida, simplemente la variable de red se suma en 1 por cada iteración, resultando en un gradiente más leve. Aún así, para hacer el filtro más notorio, creamos una variable llamada equilibrio. Equilibrio representa la cantidad de píxeles no cubiertos por la iteración (255 - píxeles_totales). De esta manera, si la imagen tiene 150 píxeles de ancho, nos faltarían 100 para completar el rojo. Para generar un equilibrio, justamente, hacemos que el rojo empiece en el valor de la mitad de los píxeles faltantes, en vez de 0. Esto resulta en un gradiente más notorio, porque si la imagen es pequeña, empezaría con rojo en cero y terminaría con rojo en 100, o similar.

Comparación Single y Multi thread

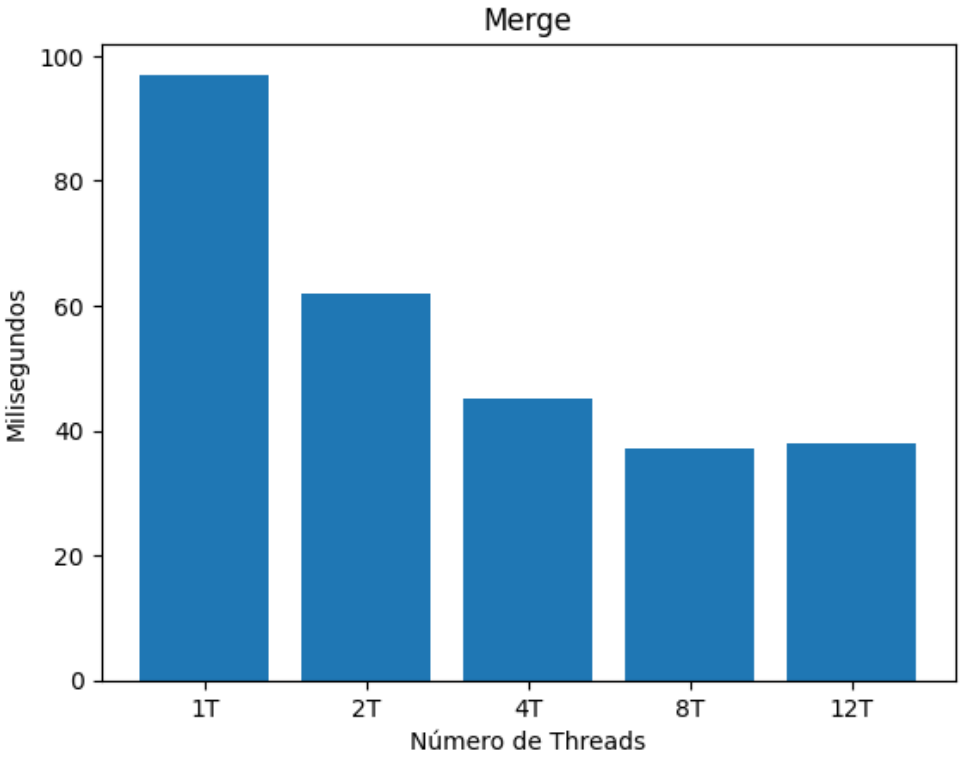
Para estas comparaciones utilizamos una imagen de “chihiro” (989 x 707). Generalmente, mientras más grande sea la imagen, mayor procesamiento realizará el filtro y el tiempo será mayor. De todas maneras, podemos enfocarnos en la proporción y tomarla como base para analizar los resultados.

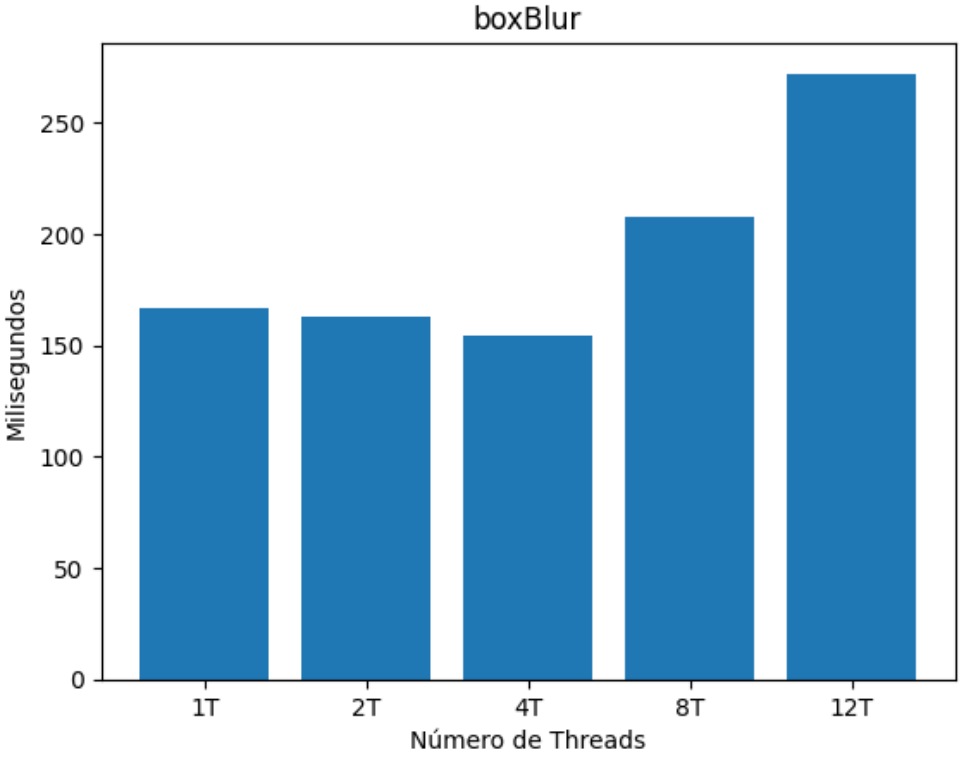
Curso: 6AO
Materia: Programación Sobre Redes
Profesor: Darío Turco
Alumnos: Federico Cerocchi, Manuel Gudiño, Matheo Maidana y Lucas Santander

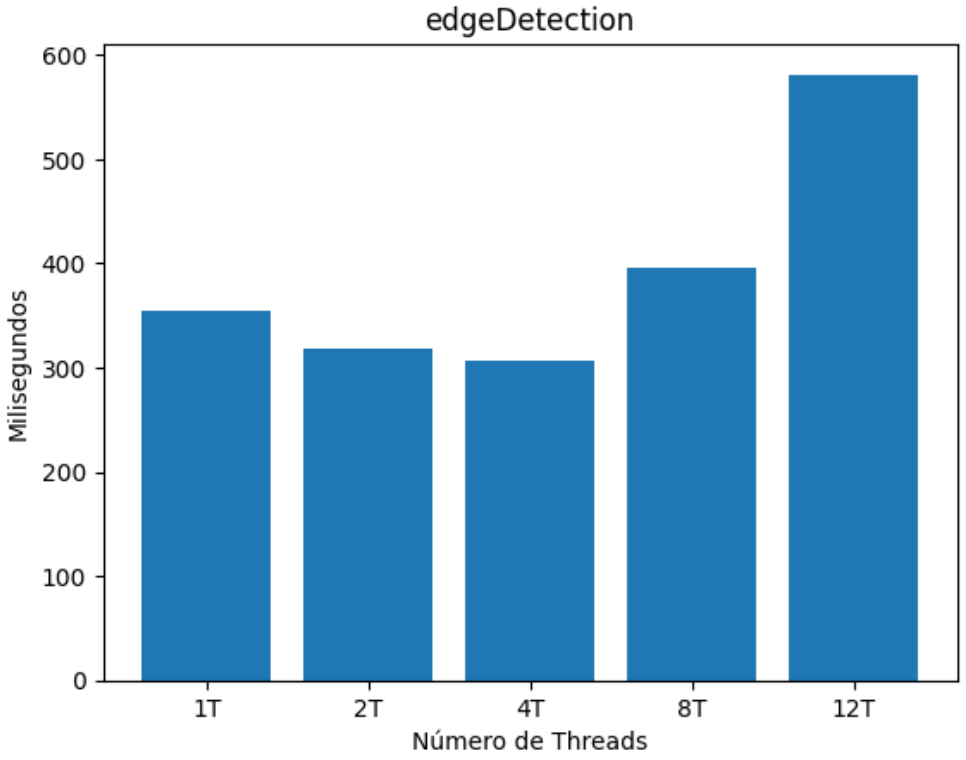




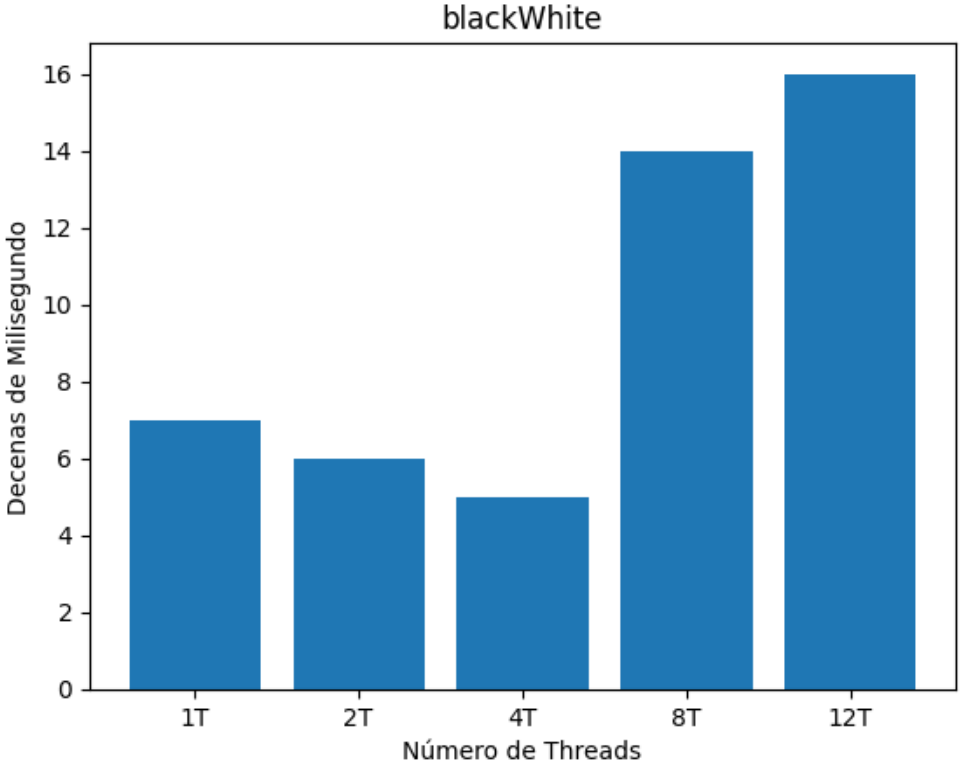
Curso: 6AO
Materia: Programación Sobre Redes
Profesor: Darío Turco
Alumnos: Federico Cerocchi, Manuel Gudiño, Matheo Maidana y Lucas Santander







Imágenes pequeñas:



Análisis

En la mayoría de filtros vimos resultados similares. La cantidad de threads es inversamente proporcional al tiempo de procesamiento. Cuantos más threads, menos segundos, y viceversa. De todas maneras, esta lógica se mantiene durante cierto número de threads. Si con 1 thread el tiempo es de 16, con 2 va a ser de 8, con 4 de 4 y con 8 de 2. Ya a partir de esa cantidad de threads, el tiempo se mantiene y va empeorando lentamente, incluso.

Contrariamente, en el caso de los filtros BoxBlur y EdgeDetection, esta proporción es directa, es decir, a mayor cantidad de threads, mayor tiempo; esta irregularidad puede deberse al kernel elegido en el filtro BoxBlur, que también se utiliza en EdgeDetection.

Estos dos párrafos, más allá de lo dicho, corresponden a cierto tipo de imágenes. En nuestro caso, una imagen considerablemente grande. Decidimos realizar un test similar sobre la imágen “house_1”, que es mucho más chicha que la imagen original. Como se puede ver en la última imagen, no tiene demasiado beneficio el multi-threading. Solo reduce el tiempo total en las primeras iteraciones (2-4 threads). Luego, el tiempo comienza a aumentar exponencialmente.

Conclusión

En conclusión, el multi-threading tiene mucho sentido en varios casos, pero con excepciones. Cuanto menos trabajo tengan que realizar los threads individualmente, menos conveniente es su uso. De utilizar más threads de los necesarios, esta práctica termina siendo contraproducente, aumentando el tiempo total. Esto se debe a que crear y correr los threads, asignarles espacio en memoria, y todo lo que su procesamiento conlleva, termina siendo superior a la carga a la que se los somete. Por esto es que a veces termina rindiendo más el single-thread.

Para ejemplificar con nuestra propia experiencia, podemos observar el primer gráfico y el último. Con una imágen de tamaño reducido, únicamente nos fue conveniente paralelizar con 2 o 4 threads. De ahí en adelante, la cosa empeoró. Con una imagen grande, pudimos sacarle provecho hasta los 12 threads. Haciendo algunos testeos más extremos, probamos con 2240 threads, y los resultados no fueron sorprendentes, el tiempo se multiplicó en un 10000%.

Es posible que los resultados obtenidos varien según el hardware sobre el cual se ejecuten las pruebas. En nuestro caso realizamos las pruebas sobre una laptop cuyos threads recomendados son 12, según el método

thread::hardware_concurrency(). Probablemente, si se realizara este test sobre un equipo súper potente, la conclusión sería la misma, pero el número de threads totales antes de ser perjudiciales sería mayor. En nuestros tests fueron 8-12 threads, como máximo, pero se podría llegar a realizar con 24, y los resultados serían diferentes.

Si queremos comparar nuestros resultados con la Ley de Amdahl, podríamos decir que coinciden en lo propuesto, pero no al 100%. Creemos que su teoría es correcta en términos generales, ya que se refiere a un cálculo matemático concreto, que tiene sentido y se cumple bajo ciertos requisitos. Pero en nuestro caso, que no tenemos recursos ilimitados, la matemática deja de ser exacta. Según la teoría, cuantos más threads el beneficio se hace más imperceptible, pero es beneficio al fin y al cabo, como si el soporte de threads fuese infinito. En los tests realizados, cuantos más threads peor performance.

También probamos incluir la flag -msse para vectorizar los algoritmos, pero no vimos muchos cambios. Los resultados fueron prácticamente los mismos tanto en los procedimientos single como en los multi-thread.

Con respecto al loader, pudimos implementar esta nueva funcionalidad en ambos casos. Aún así, los resultados en multi-thread no fueron demasiado positivos. Esto se debe a que al realizar los mismos procedimientos multi-thread para cada imagen, estamos llamando a múltiples threads para cada una de ellas. Es decir, si colocamos 8 threads para aplicar el filtro individualmente en todas las imágenes del directorio, estaríamos creando (8 * cantidad_imagenes) threads en total y en simultáneo. Teniendo esto en cuenta, son realmente pocas las posibilidades que tenemos para hacer al multi-threading efectivo en estos casos. Cuantas menos imágenes y menos threads se creen, probablemente los resultados sean mejores.

Una alternativa a esta manera de aplicar los filtros sería hacer un thread por cada imagen, pero estaríamos quitando al usuario la posibilidad de elegir su cantidad de threads. Además, seguramente no se tenga en cuenta lo mencionado en el párrafo anterior, ya que para identificarlo se debe profundizar en el funcionamiento técnico del código. Al fin y al cabo, es más efectivo el loader single-thread.

¿Por qué línea seguimos?

Para el experimento teníamos pensado aplicar los filtros en gifs, pero nos resultó muy complejo y lo dejamos de lado. Quizás en un futuro lo retomemos y volvamos a indagar sobre el tema.